gpasm 0.0.7

James Bowman

Dec 2 1998

Contents

Introduction

gpasm is meant to be an open source replacement for the Microchip (TM) product MPASM, an assembler for Microchip's popular PICmicro (TM) line of microcontrollers. This manual covers the basics of running gpasm: for more details on a microprocessor, consult the manual for the specific PICmicro product that you are using.

This document is part of gpasm.

Chapter 1

Running gpasm

The general syntax for running gpasm is

```
gpasm <options> <asm-file>
```

Where options can be one of:

Option	Meaning
a <format>	Produce hex file in one of three formats: inhx8m (the default), inhx8s, inhx32.
d symbol[=value]	Equivalent to "#define <symbol> <value>".
c	Do not ignore case in source code. By default gpasms to treat "fooYa" and "FOOYA"as equal.
p <processor>	Select target processor, which may be one of the processors listed in section 4.1.
r <radix>	Set the radix, i.e. the number base that gpasm uses when interpreting numbers. <radix> can be one of "oct", "dec" and "hex" for bases eight, ten, and sixteen respectively. Default is "hex".
v	Print gpasm version information and exit.

Unless otherwise specified, gpasm removes the ".asm" suffix from its input file, replacing it with ".lst" and ".hex" for the list and hex output files respectively. On most modern operating systems case is significant in filenames. For this reason you should ensure that filenames are named consistently, and that the ".asm" suffix on any source file is in lower case.

gpasm always produces a ".lst" file. If it runs without errors, it also produces a ".hex" file.

1.1 Using gpasm with "make"

On most operating systems, you can build a project using the make utility. To use gpasm with make, you might have a "makefile" like this:

```
tree.hex: tree.asm treedef.inc
        gpasm tree.asm
```

This will rebuild “tree.hex” whenever either of the “tree.asm” or “treedef.inc” files change.

1.2 Dealing with errors

gpasm doesn’t specifically create an error file. This can be a problem if you want to keep a record of errors, or if your assembly produces so many errors that they scroll off the screen. To deal with this if your shell is “sh”, “bash” or “ksh”, you can do something like:

```
gpasm tree.asm 2>&1 | tee tree.err
```

This redirects standard error to standard output (“2>&1”), then pipes this output into “tee”, which copies it input to “tree.err”, and then displays it.

Chapter 2

Syntax

2.1 File structure

gpasm source files consist of a series of lines. Lines can contain a label (starting in column 1) or an operation (starting in any column after 1), both, or neither. Comments follow a ";" character, and are treated as a newline. Labels may be any series of the letters A-z, digits 0-9, and the underscore ("_"); they may not begin with a digit. Labels may be followed by a colon (":").

An operation is a single identifier (the same rules as for a label above) followed by a space, and a comma-separated list of parameters. For example, the following are all legal source lines:

```
                                   ; Blank line
loop    sleep                      ; Label and operation
        incf    6,1                ; Operation with 2 parameters
        goto    loop               ; Operation with 1 parameter
```

2.2 Expressions

gpasm supports a full set of operators, based on the C operator set. The operators in the following table are arranged in groups of equal precedence, but the groups are arranged in order of increasing precedence. When gpasm encounters operators of equal precedence, it always evaluates from left to right.

Operator	Description
=	assignment
&&	logical and
||	logical or
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<	less than
>	greater than
==	equals
!=	not equals
>=	greater than or equal
<=	less than or equal
<<	left shift
>>	right shift
+	addition
-	subtraction
*	multiplication
/	division
%	modulo
HIGH	high byte
LOW	low byte
-	negation
!	logical not
~	bitwise not

2.3 Numbers

gpasm gives you several ways of specifying numbers. You can use a syntax that uses an initial character to indicate the number's base. The following table summarizes the alternatives. Note the C-style option for specifying hexadecimal numbers.

base	general syntax	21 decimal written as
binary	B'[01]*'	B'10101'
octal	O'[0-7]*'	O'25'
decimal	D'[0-9]*'	D'21'
hex	H'[0-F]*'	H'15'
hex	0x[0-F]*	0x15

When you write a number without a specifying prefix such as "45", gpasm uses the current radix (base) to interpret the number. You can change this radix with the RADIX directive, or with the "-r" option on gpasm's command-line. If you do not start hexadecimal numbers with a digit, gpasm will attempt to interpret what you've written as an identifier. For example, instead of writing C2, write either 0C2, 0xC2 or H'C2'.

Case is not significant when interpreting numbers: 0ca, 0CA, h'CA' and H'ca' are all equivalent.

You can write the ASCII code for a character X using 'X', or A'X'.

2.4 Preprocessor

A line such as:

```
include foo.inc
```

will make gpasm fetch source lines from the file "foo.inc" until the end of the file, and then return to the original source file at the line following the include.

Lines beginning with a "#" are preprocessor directives, and are treated differently by gpasm. They may contain a "#define", or a "#undefine" directive.

Once gpasm has processed a line such as:

```
#define X Y
```

every subsequent occurrence of X is replaced with Y, until the end of file or a line

```
#undefine X
```

appears.

The preprocessor in gpasm is only *like* the C preprocessor; its syntax is rather different from that of the C preprocessor. gpasm uses a simple internal preprocessor to implement "include", "#define" and "#undefine".

Chapter 3

Directives

3.1 Code generation

To set the PIC memory location where gpasm will start assembling code, use the ORG directive. If you don't specify an address with ORG, gpasm assumes 0x0000.

3.2 Configuration

You can choose the fuse settings for your PIC implementation using the __CONFIG directive, so that the hex file set the fuses explicitly. Naturally you should make sure that these settings match your PIC hardware design.

The __MAXRAM and __BADRAM directives specify which RAM locations are legal. These directives are mostly used in processor-specific configuration files.

3.3 Conditional assembly

The IF, IFNDEF, IFDEF, ELSE and ENDIF directives enable you to assemble certain sections of code only if a condition is met. In themselves, they do not cause gpasm to emit any PIC code. The example in section 3.4 for demonstrates conditional assembly.

3.4 Macros

gpasm supports a simple macro scheme; you can define and use macros like this:

```
any     macro parm
        movlw parm
        endm
...
        any   33
```

A more useful example of some macros in use is:

```
; Shift reg left, result (w or f) in 'dst'
slf     macro   reg,dst
        clrc
        rlf     reg,f
        endm

; Scale W by "factor".  Result in "reg", W unchanged.
scale   macro   reg, factor
        if (factor == 1)
            movwf reg                   ; multiplication by 1 is easy
        else
            scale   reg, (factor / 2) ; reg = W * (factor / 2)
            slf     reg,f               ; double reg
            if ((factor & 1) == 1)      ; if lo-bit of factor is 1 ..
                addwf   reg,f           ; .. add W to reg
            endif
        endif
        endm
```

This recursive macro generates code to multiply W by a constant "factor", and stores the result in "reg". So writing:

```
        scale   tmp,D'10'
```

is the same as writing:

```
        movwf   tmp                     ; tmp = W
        clrc
        rlf     tmp,f                   ; tmp = 2 * W
        clrc
        rlf     tmp,f                   ; tmp = 4 * W
        addwf   tmp,f                   ; tmp = (4 * W) + W = 5 * W
        clrc
        rlf     tmp,f                   ; tmp = 10 * W
```

3.5 Suggestions for structuring your code

Nested IF operations can quickly become confusing. Indentation is one way of making code clearer. Another way is to add braces on IF, ELSE and ENDIF, like this:

```
        IF (this) ; {
          ...
        ELSE      ; }{
          ...
        ENDIF     ; }
```

After you've done this, you can use your text editor's show-matching-brace to check matching parts of the IF structure. In vi this command is "%", in emacs it's M-C-f and M-C-b.

3.6 Directive summary

__BADRAM

```
__BADRAM <expression> [, <expression]*
```

instructs gpasm that it should generate an error if there is any use of the given RAM locations. Specify a range of addresses with <lo>-<hi>. See any processor-specific header file for an example.

See also: __MAXRAM

__CONFIG

```
__CONFIG <expression>
```

sets the PIC processor's configuration fuses.

__MAXRAM

```
__MAXRAM <expression>
```

instructs gpasm that an attempt to use any RAM location above the one specified should be treated as an error. See any processor specific header file for an example.

See also: __BADRAM

CBLOCK

```
CBLOCK <expression>
```

Marks the beginning of a block of constants. gpasm allocates values for symbols in the block starting at the value given to CBLOCK.

See also: EQU

DATA

```
DATA <expression> [, <expression]*
```

generates the specified data.

See also: DT

DT

```
DT <expression> [, <expression>]*
```

generates the specified data as bytes in a sequence of RETLW instructions.
See also: DATA

ELSE

```
ELSE
```

marks the alternate section of a conditional assembly block.
See also: IF, IFDEF, IFNDEF, ELSE, ENDIF

END

```
END
```

marks the end of the source file.

ENDC

```
ENDC
```

marks the end of a CBLOCK.
See also: CBLOCK

ENDIF

```
ENDIF
```

ends a conditional assembly block.
See also: IFDEF, IFNDEF, ELSE, ENDIF

ENDM

```
ENDM
```

ends a macro definition.
See also: MACRO

EQU

```
<label> EQU <expression>
```

permanently assigns the value obtained by evaluating <expression> to the symbol <label>.
See also: SET

IF

```
IF <expression>
```

begin a conditional assembly block. If the value obtained by evaluating <expression> is true (i.e. non-zero), code up to the following ELSE or ENDIF is assembled. If the value is false (i.e. zero), code is not assembled until the corresponding ELSE or ENDIF.

See also: IFDEF, IFNDEF, ELSE, ENDIF

IFDEF

```
IFDEF <symbol>
```

begin a conditional assembly block. If <symbol> appeared previously in a '#define' directive, gpasm assembles the following code.

See also: IF, IFNDEF, ELSE, ENDIF

IFNDEF

```
IFNDEF <symbol>
```

begin a conditional assembly block. If <symbol>has not appeared previously in a '#define' directive, gpasm assembles the following code.

See also: IF, IFNDEF, ELSE, ENDIF

LIST

```
LIST <expression> [ , <expression> ] *
```

enables output to the list (".lst") file. It may also change the state of gpasm in various surprising ways:

<table>
<tr><th>option</th><th>description</th></tr>
<tr><td>n=<expression></td><td>Sets the number of lines per page</td></tr>
<tr><td>r= [oct | dec | hex]</td><td>Sets the radix</td></tr>
<tr><td>st = [ON | OFF]</td><td>Symbol table dump on or off</td></tr>
<tr><td>p = <symbol></td><td>Sets the current processor</td></tr>
</table>

See also: NOLIST, RADIX, PROCESSOR

LOCAL

```
LOCAL <symbol> [ , <symbol> ]
```

declares <symbol> as local to the macro that's currently being defined. This means that further occurrences of <symbol> in the macro definition refer to a local variable, with scope and lifetime limited to the execution of the macro.

See also: MACRO, ENDM

MACRO

```
<label> MACRO [ <symbol> [ , <symbol> ]* ]
```

declares a macro with name <label>. gpasm replaces any occurrences of <symbol> in the macro definition with the parameters given at macro invocation.

See also: LOCAL, ENDM

MESSG

```
MESSG <string>
```

writes <string> to the list file, and to the standard error output.

NOLIST

```
NOLIST
```

disables list file output.

See also: LIST

ORG

```
ORG <expression>
```

sets the location at which instructions will be placed. If the source file does not specify an address with ORG, gpasm assumes an ORG of zero.

PAGE

```
PAGE
```

causes the list file to advance to the next page.

See also: LIST

PROCESSOR

```
PROCESSOR <symbol>
```

selects the target processor. See section 4.1 for more details.

See also: LIST

RADIX

```
RADIX <symbol>
```

selects the default radix from “oct” for octal, “dec” for decimal or “hex” for hexadecimal. gpasm uses this radix to interpret numbers that don’t have an explicit radix.

See also: LIST

SPACE

```
SPACE <expression>
```

sets aside <expression> words in the code area. SPACE does this by advancing the code location by <expression>.

See also: ORG

Chapter 4

Instructions

4.1 Supported processors

gpasm currently supports the following processors:

- PIC12C508
- PIC12C509
- PIC16C54
- PIC16C55
- PIC16C56
- PIC16C57
- PIC16C58
- PIC16C71
- PIC16C84
- PIC16F84
- SX18
- SX20
- SX28

The instruction set that gpasm supports depends on the selected processor.

4.2 Instruction set summary

For 12-bit Microchip parts (12C5XX), the assembler supports the following instructions.

Syntax	Description
ADDWF <f>,<dst>	Add W to <f>, result in <dst>
ANDLW <f>,<dst>	And W and literal, result in W
ANDWF <f>,<dst>	And W and <f>, result in <dst>
BCF <f>,<bit>	Clear <bit> of <f>
BSF <f>,<bit>	Set <bit> of <f>
BTFSC <f>,<bit>	Skip next instruction if <bit> of <f> is clear
BTFSS <f>,<bit>	Skip next instruction if <bit> of <f> is set
CALL <addr>	Call subroutine
CLRF <f>,<dst>	Write zero to <dst>
CLRW	Write zero to W
CLRWDT	Reset watchdog timer
COMF <f>,<dst>	Complement <f>, result in <dst>
DECF <f>,<dst>	Decrement <f>, result in <dst>
DECFSZ <f>,<dst>	Decrement <f>, result in <dst>, skip if zero
GOTO <addr>	Go to <addr>
INCF <f>,<dst>	Increment <f>, result in <dst>
INCFSZ <f>,<dst>	Increment <f>, result in <dst>, skip if zero
IORLW <f>,<dst>	Or W and <f>, result in <dst>
MOVF <f>,<dst>	Move <f> to <dst>
MOVLW <imm8>	Move literal to W
MOVWF <f>	Move W to <f>
NOP	No operation
OPTION	
RETLW <imm8>	Load W with immediate and return
RLF <f>,<dst>	Rotate <f> left, result in <dst>
RRF <f>,<dst>	Rotate <f> right, result in <dst>
SLEEP	Enter sleep mode
SUBWF <f>,<dst>	Subtract W from <f>, result in <dst>
SWAPF <f>,<dst>	Swap nibbles of <f>, result in <dst>
TRIS	
XORLW	Xor W and <f>, result in <dst>
XORWF	Xor W and immediate

For 14-bit Microchip parts (16CXX), the assembler supports the following instructions.

Syntax	Description
ADDLW <imm8>	Add immediate to W
ADDWF <f>,<dst>	Add W to <f>, result in <dst>
ANDWF <f>,<dst>	And W and <f>, result in <dst>
BCF <f>,<bit>	Clear <bit> of <f>
BSF <f>,<bit>	Set <bit> of <f>
BTFSC <f>,<bit>	Skip next instruction if <bit> of <f> is clear
BTFSS <f>,<bit>	Skip next instruction if <bit> of <f> is set
CALL <addr>	Call subroutine
CLRF <f>,<dst>	Write zero to <dst>
CLRW	Write zero to W
CLRWDT	Reset watchdog timer
COMF <f>,<dst>	Complement <f>, result in <dst>
DECF <f>,<dst>	Decrement <f>, result in <dst>
DECFSZ <f>,<dst>	Decrement <f>, result in <dst>, skip if zero
GOTO <addr>	Go to <addr>
INCF <f>,<dst>	Increment <f>, result in <dst>
INCFSZ <f>,<dst>	Increment <f>, result in <dst>, skip if zero
IORLW <f>,<dst>	Or W and <f>, result in <dst>
MOVF <f>,<dst>	Move <f> to <dst>
MOVLW <imm8>	Move literal to W
MOVWF <f>	Move W to <f>
NOP	No operation
OPTION	
RETFIE	Return from interrupt
RETLW <imm8>	Load W with immediate and return
RETURN	Return from subroutine
RLF <f>,<dst>	Rotate <f> left, result in <dst>
RRF <f>,<dst>	Rotate <f> right, result in <dst>
SLEEP	Enter sleep mode
SUBLW	Subtract W from literal
SUBWF <f>,<dst>	Subtract W from <f>, result in <dst>
SWAPF <f>,<dst>	Swap nibbles of <f>, result in <dst>
TRIS	
XORLW	Xor W and <f>, result in <dst>
XORWF	Xor W and immediate

For Scenix processors, the assembler supports the following instructions, in addition to those listed under "12-bit processors" above.

Syntax	Description
BANK <imm3>	
IREAD	
MODE <imm4>	
MOVMW	
MOVWM	
PAGE <imm3>	
RETI	
RETIW	
RETP	
RETURN	

There are also a number of standard additional macros that gpasm reads from the file “special.inc”. Consult that file for more details. These macros are:

Syntax	Description
ADDCF <f>,<dst>	Add carry to <f>, result in <dst>
B <addr>	Branch
BC <addr>	Branch on carry
BZ <addr>	Branch on zero
BNC <addr>	Branch on no carry
BNZ <addr>	Branch on not zero
CLRC	Clear carry
CLRZ	Clear zero
SETC	Set carry
SETZ	Set zero
MOVFW <f>	Move file to W
NEGF <f>	Negate <f>
SKPC	Skip on carry
SKPZ	Skip on zero
SKPNC	Skip on no carry
SKPNZ	Skip on not zero
SUBCF <f>,<dst>	Subtract carry from <f>, result in <dst>
TSTF <f>	Test <f>

4.3 Instruction set for PIC16CXX

Syntax	Description
ADDLW <imm8>	Add immediate to W
ADDWF <f>,<dst>	Add W to <f>, result in <dst>
ANDLW <f>,<dst>	And W and <f>, result in <dst>
BCF <f>,<bit>	Clear <bit> of <f>
BSF <f>,<bit>	Set <bit> of <f>
BTFSC <f>,<bit>	Skip next instruction if <bit> of <f> is clear
BTFSS <f>,<bit>	Skip next instruction if <bit> of <f> is set
CALL <addr>	Call subroutine
CLRF <f>,<dst>	Write zero to <dst>
CLRW	Write zero to W
CLRWDT	Reset watchdog timer
COMF <f>,<dst>	Complement <f>, result in <dst>
DECF <f>,<dst>	Decrement <f>, result in <dst>
DECFSZ <f>,<dst>	Decrement <f>, result in <dst>, skip if zero
GOTO <addr>	Go to <addr>
INCF <f>,<dst>	Increment <f>, result in <dst>
INCFSZ <f>,<dst>	Increment <f>, result in <dst>, skip if zero
IORLW <f>,<dst>	Or W and <f>, result in <dst>
MOVF <f>,<dst>	Move <f> to <dst>
MOVLW <imm8>	Move literal to W
MOVWF <f>	Move W to <f>
NOP	No operation
OPTION	
RETFIE	Return from interrupt
RETLW <imm8>	Load W with immediate and return
RETURN	Return from subroutine
RLF <f>,<dst>	Rotate <f> left, result in <dst>
RRF <f>,<dst>	Rotate <f> right, result in <dst>
SLEEP	Enter sleep mode
SUBLW	Subtract W from literal
SUBWF <f>,<dst>	Subtract W from <f>, result in <dst>
SWAPF <f>,<dst>	Swap nibbles of <f>, result in <dst>
TRIS	
XORLW	Xor W and <f>, result in <dst>
XORWF	Xor W and immediate

There are also a number of standard additional macros that gpasm reads from the file "special.inc". Consult that file for more details. These macros are:

Syntax	Description
ADDCF <f>,<dst>	Add carry to <f>, result in <dst>
B <addr>	Branch
BC <addr>	Branch on carry
BZ <addr>	Branch on zero
BNC <addr>	Branch on no carry
BNZ <addr>	Branch on not zero
CLRC	Clear carry
CLRZ	Clear zero
SETC	Set carry
SETZ	Set zero
MOVFW <f>	Move file to W
NEGF <f>	Negate <f>
SKPC	Skip on carry
SKPZ	Skip on zero
SKPNC	Skip on no carry
SKPNZ	Skip on not zero
SUBCF <f>,<dst>	Subtract carry from <f>, result in <dst>
TSTF <f>	Test <f>

Chapter 5

Errors

gpasm writes every error message to two locations:

- the standard error output
- the list file (".lst")

The format of error messages is:

```
Error <src-file> <line> : <code> <description>
```

where:

<src-file> is the source file where gpasm encountered the error

<line> is the line number

<code> is the 3-digit code for the error, given in the list below

<description> is a short description of the error. In some cases this contains further information about the error.

Error messages are suitable for parsing by emacs' "compilation mode".

This chapter lists the error messages that gpasm produces.

101 ERROR directive

A user-generated error. See the ERROR directive for more details.

102 Out of memory

gpasm has run out of memory: check your operating system's settings to determine how much virtual memory is available. If there's plenty of memory available, this error probably means that there was a bug in gpasm.

103 Syntax error

Check the source code at the given line for syntax. Ensure that labels start in column 1, parentheses are balanced, and numbers follow the rules in section 2.3.

104 Can't evaluate expression

gpasm was unable to determine the value of an expression. This usually means that a label was unresolved.

105 Too few/many operands for opcode, expected X

If the operation was a machine instruction, check the opcode in chapter 4. If the operation was a macro definition, check the macro's definition in the source.

106 Invalid RAM address

A __MAXRAM or __BADRAM operation specified a RAM address that was out of range.

107 Unexpected "else"

gpasm encountered an “else” directive that didn't seem to have a preceding “if”. See the suggestions in section 3.5.

108 Unexpected "endif"

gpasm encountered an “endif” directive that didn't seem to have a preceding “if” or “else”. See the suggestions in section 3.5.

109 Expression too complex

gpasm encountered a complex expression where it expected a simple one, usually a single identifier. Some directives that expect simple parameters: MACRO, IFDEF, IFNDEF, LOCAL, LIST, PROCESSOR, RADIX.

110 Expected string

The MESSG directive only accepts a single string argument.

112 Unknown opcode "X"

The opcode is not a known instruction for the selected processor, nor is it a defined macro. Check the spelling of the opcode, and the selected processor. If it's a macro, check the macro definition.

113 Illegal character X in numeric constant

Legal characters in number constants are 0-1 for binary, 0-8 for octal, 0-9 for decimal, and 0-F for hexadecimal.

114 Value of symbol "X" differs on second pass gpasm is a two-pass assembler. If the value of a symbol is different on the first and second passes, there is a problem with the code.

115 Unable to open file "X"

The file X could not be found. Check that it is in the include path. Filenames are case-sensitive on most modern operating systems.

116 Unexpected “endm”

gpasm encountered an “endm” directive that didn’t seem to have a preceding “macro”. Check that the macro wasn’t already ended.

117 Missing macro label

Macros must have a label.

118 Attempt to redefine macro; first definition at line X

A second attempt was made to define the same macro.

119 Attempt to use "local" outside macro definition

The local directive is only meaningful inside a macro definition. See section 3.4.

120 Duplicate macro parameter

You cannot use the same identifier more than one in a macro definition’s parameter list.

121 Processor already selected

A processor has already been selected. You must select the processor only once.

122 Unrecognized processor

The specified processor is not recognized by gpasm. Check the processors listed in section 4.1.

123 Unrecognized radix

A radix specified on the command-line or in the RADIX directive was invalid. Choices are OCT, DEC and HEX for octal, decimal and hexadecimal respectively.

125 Option should be "OFF" or "ON"

See the LIST directive in section 3.6.

126 Attempt to use illegal RAM location

The instruction referenced a RAM location specified as illegal in a BADRAM or MAXRAM directive.

127 Bit address out of range

The instruction specified a bit position that is out of range. Valid range is 0 (least significant bit) to 7 (most significant bit).

Index